Pieter-Jan.com
Creativity In Automation & More

Reading a IMU Without Kalman: The Complementary Filter

Submitted by Pieter-Jan on Fri, 26/04/2013 - 08:38

These days, IMU's (Intertial Measurement Units) are used everywhere. They are e.g. the sensors that are responsible for keeping track of the oriëntation of your mobile phone. This can be very useful for automatic screen tilting etc. The reason I am interested in this sensor is because I want to use it to stabilize my quadrocopter. If you are here for another reason, this is not a problem as this tutorial will apply for everyone.

When looking for the best way to make use of a IMU-sensor, thus **combine the accelerometer and gyroscope data**, a lot of people get fooled into using the very powerful but complex Kalman filter. However the Kalman filter is great, there are 2 big problems with it that make it hard to use:

- Very complex to understand.
- Very hard, if not impossible, to implement on certain hardware (8-bit microcontroller etc.)

In this tutorial I will present a solution for both of these problems with another type of filter: the complementary filter. It's extremely easy to understand, and even easier to implement.

Why do I need a filter?

Most IMU's have 6 DOF (Degrees Of Freedom). This means that there are 3 accelerometers, and 3 gyrosocopes inside the unit. If you remember anything from a robotics class you might have taken, you might be fooled into thinking that the IMU will be able to measure the precise position and orientation of the object it is attached to. This because they have told you that an object in free space has 6DOF. So if we can measure them all, we know everything, right? Well ... not really. The sensor data is not good enough to be used in this way.

We will use both the accelerometer and gyroscope data for the same purpose: obtaining the angular position of the object. The gyroscope can do this by integrating the angular velocity over time, as was explained in a previous article. To obtain the angular position with the accelerometer, we are going to determine the position of the gravity vector (g-force) which is always visible on the accelerometer. This can easily be done by using an atan2 function. In both these cases, there is a big problem, which makes the data very hard to use without filter.

The problem with accelerometers

As an accelerometer measures all forces that are working on the object, it will also see a lot more than just the gravity vector. Every small force working on the object will disturb our measurement completely. If we are working on an actuated system (like the quadrocopter), then the forces that drive the system will be visible on the sensor as well. The accelerometer data is reliable only on the long term, so a "low pass" filter has to be used.

The problem with gyroscopes

In one of the previous articles I explained how to obtain the angular position by use of a gyroscope. We saw that it was very easy to obtain an accurate measurement that was not susceptible to external forces. The less good news was that, because of the integration over time, the measurement has the tendency to drift, not returning to zero when the system went back to its original position. The gyroscope data is reliable only on the short term, as it starts to drift on the long term.

The complementary filter

The complementary filter gives us a "best of both worlds" kind of deal. On the short term, we use the data from the gyroscope, because it is very precise and not susceptible to external forces. On the long term, we use the data from the accelerometer, as it does not drift. In it's most simple form, the filter looks as follows:

$$angle = 0.98 * (angle + gyrData * dt) + 0.02 * (accData)$$

The gyroscope data is integrated every timestep with the current angle value. After this it is combined with the low-pass data from the

accelerometer (already processed with atan2). The constants (0.98 and 0.02) have to add up to 1 but can of course be changed to tune the filter properly.

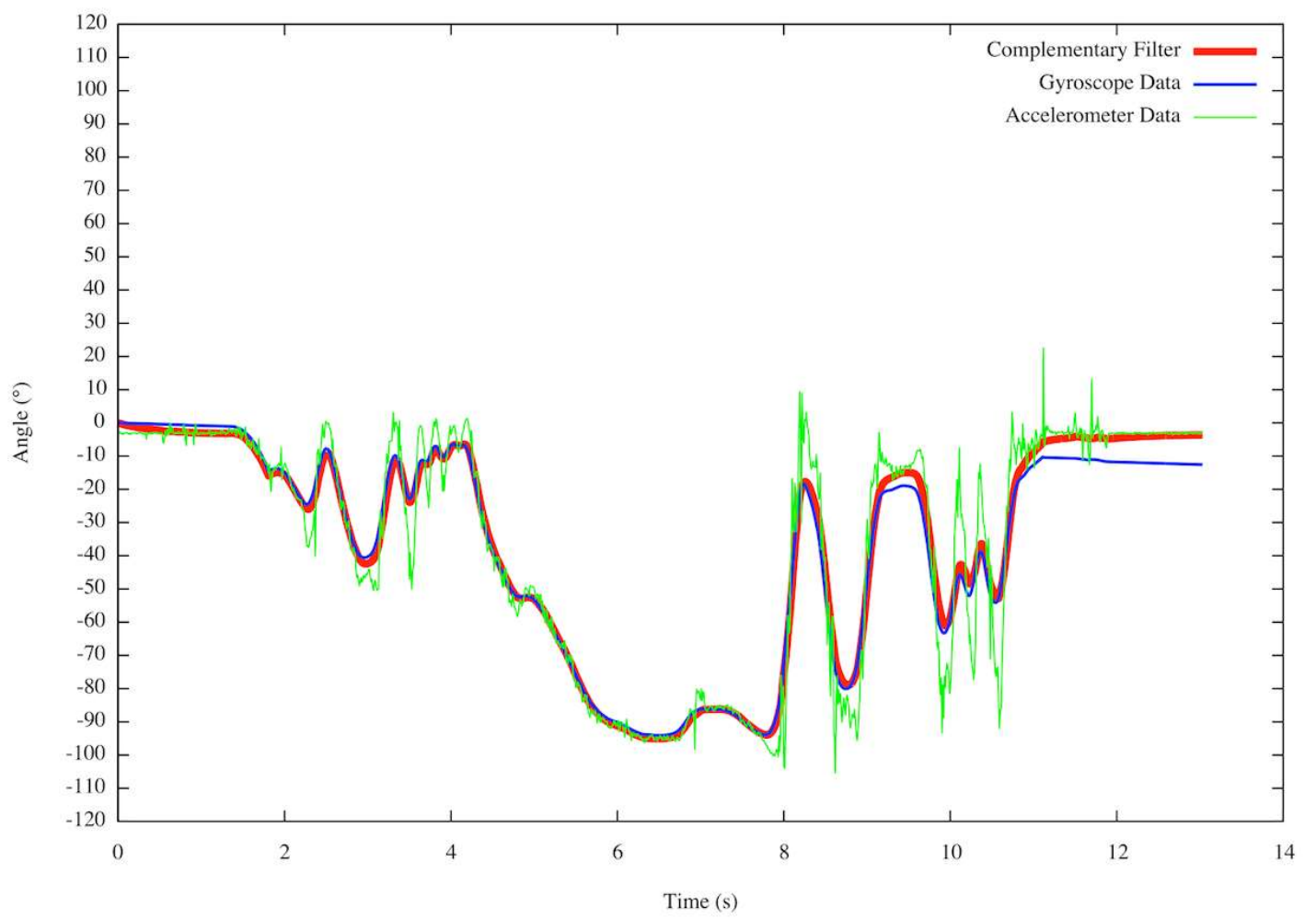
I implemented this filter on a Raspberry Pi using a MPU6050 IMU. I will not discuss how to read data from the MPU6050 in this article (contact me if you want the source code). The implementation of the filter is shown in the code snippet below. As you can see it is very easy in comparison to Kalman.

The function "ComplementaryFilter" has to be used in a infinite loop. Every iteration the pitch and roll angle values are updated with the new gyroscope values by means of integration over time. The filter then checks if the magnitude of the force seen by the accelerometer has a reasonable value that could be the real g-force vector. If the value is too small or too big, we know for sure that it is a disturbance we don't need to take into account. Afterwards, it will update the pitch and roll angles with the accelerometer data by taking 98% of the current value, and adding 2% of the angle calculated by the accelerometer. This will ensure that the measurement won't drift, but that it will be very accurate on the short term.

It should be noted that this code snippet is only an example, and should not be copy pasted as it will probably not work like that without using the exact same settings as me.

```
#define ACCELEROMETER_SENSITIVITY 8192.0
#define GYROSCOPE_SENSITIVITY 65.536

#define M_PI 3.14159265359

#define dt 0.01                                         // 10 ms sample rate!

void ComplementaryFilter(short accData[3], short gyrData[3], float *pitch, float *roll)
{
    float pitchAcc, rollAcc;

    // Integrate the gyroscope data -> int(angularSpeed) = angle
    *pitch += ((float)gyrData[0] / GYROSCOPE_SENSITIVITY) * dt; // Angle around the X-axis
    *roll -= ((float)gyrData[1] / GYROSCOPE_SENSITIVITY) * dt;    // Angle around the Y-axis

    // Compensate for drift with accelerometer data if !bullshit
    // Sensitivity = -2 to 2 G at 16Bit -> 2G = 32768 && 0.5G = 8192
    int forceMagnitudeApprox = abs(accData[0]) + abs(accData[1]) + abs(accData[2]);
    if (forceMagnitudeApprox > 8192 && forceMagnitudeApprox < 32768)
    {
        // Turning around the X axis results in a vector on the Y-axis
        pitchAcc = atan2f((float)accData[1], (float)accData[2]) * 180 / M_PI;
        *pitch = *pitch * 0.98 + pitchAcc * 0.02;

        // Turning around the Y axis results in a vector on the X-axis
        rollAcc = atan2f((float)accData[0], (float)accData[2]) * 180 / M_PI;
        *roll = *roll * 0.98 + rollAcc * 0.02;
    }
}
```

If we use the sweetness of having a **real** computer (Raspberry Pi) collecting our data, we can easily create a graph using GNUPlot. It's easily seen that the filter (red) follows the gyroscope (blue) for fast changes, but keeps following the mean value of the accelerometer (green) for slower changes, thus not feeling the noisy accelerometer data and not drifting away eiter.



The filter is very easy and light to implement making it perfect for embedded systems. It should be noted that for stabilization systems (like the quadrocopter), the angle will never be very big. The atan2 function can then be approximated by using a small angle approximation. In this way, this filter could easily fit on an 8 bit system.

Hope this article was of any help.

Tags:
Complementary Filter Gyroscope Accelerometer IMU Quadrocopter